大作业报告

袁晨圃，李知谦，邱子陶

完整代码：https://github.com/cauphenuny/data-structure-assignment

一、 实习 2.4 马踏棋盘问题演示

1.1 算法

1.1.1 暴力搜索算法

暴力算法实现 `solve_brute_force(Point start)`：

通过手动维护一个栈记录走过的路径结点坐标与正在尝试的方向，当无路可走时利用栈进行回溯从而尝试新的路径。

```
class SimpleStack {
private:
    T* base;
    int top;
    int capacity;  // 当前容量

    void expand() { ...
public:
    SimpleStack(int initial_capacity = 100) : base(nullptr), top(-1),
capacity(initial_capacity)    { ...                               // 构造
    ~SimpleStack() { ...             // 销毁
    void push(const T& value) { ... // 入栈
    void pop() { ...                 // 出栈
    T& peek() { ...                  // 读栈顶
    bool empty() const { ...         // 判断是否栈空
    int size() const { ...           // 返回栈元素个数
```

```
struct Node {
    Point pos;   // 当前坐标
    int move_index;  // 当前正在尝试的方向
};
```

—

为了可视化搜索的过程，我们决定记录过程中每一步试探（包括回溯）。

最开始算法中每步都存储完整的棋盘（二维数组），但每步都存储一个棋盘带来的内存占用过大。

后来我们决定只使用起始点与终止点的坐标对记录每一步的行动，通过 `stepNext` 标签记录该步是前进还是回溯。

```
struct Arrow {
    Point start, end;
    bool stepNext; // 前进为1，后退为0
};

void Print_board(Board);

using Path = std::vector<Arrow>; // 一条可行路径

// 算法返回值：
return std::vector<Path> //允许返回多条可行路径
```

但即便如此，暴力搜索算法的大量路径试探仍会带来无法承受的内存开销。

—

1.1.2 贪心算法

贪心算法实现 solve_heuristic(Point start)：

基于 H. C. von Warnsdorf 于 1823 提出的 Warnsdorf's Rule ——每步选择可移动方向最少得位置移动。该算法可以以极快的速度给出一个可行解。

通过 count_onward_moves() 计算落点的可走步数：

```
static int count_onward_moves(const Board& board, int x, int y) {
    int count = 0;
    for (int i = 0; i < 8; ++i) {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (nx >= 0 && nx < BOARD_SIZE && ny >= 0 && ny < BOARD_SIZE && board(nx, ny) ==
0) {
            ++count;
        }
    }
    return count;
}
```

—

在 solve_heuristic(Point start) 内部对每一步的 MoveOption 数组排序，并向最小的方向进发。

```
// 枚举所有下一步的可选走法
for (int i = 0; i < 8; ++i) {
    int nx = x + dx[i];
    int ny = y + dy[i];
    if (nx >= 0 && nx < BOARD_SIZE && ny >= 0 && ny < BOARD_SIZE && board(nx, ny) == 0) {
        int onward = count_onward_moves(board, nx, ny);
        options.push_back({nx, ny, onward});
    }
}
// 按后继步数升序排序
std::sort(options.begin(), options.end(), [](const MoveOption& a, const MoveOption& b) {
    return a.onward < b.onward;
});
```

但贪心算法只能得到一个可行解。我们希望算法可以找到多条可行解（具有找到全部可行解的潜力）。

—

1.1.3 基于 Warnsdorf's Rule 的深度搜索算法

算法实现 solve_heuristic_enhancer(Point start)：

首次到达某结点时，基于 Warnsdorf's Rule 对其可行方向排序，优先选择出路最少的方向从而减少回溯次数。

由于框架仍是深度优先搜索，如果想找到多条可行路径，只需在找到一条可行路径后继续回溯即可。

```
while (!stk.empty()) {
    if (step == BOARD_SIZE * BOARD_SIZE) {
        if (!stk.empty()) {...} // 保存最后一步结果
        if(countHistory == NUM_OF_PATH) break;
        else {// 后退一步继续搜其他路径
            Point end = stk.peek().pos;
            stk.pop();  --step; board(end.x, end.y) = 0;
            Point start = stk.empty() ? end : stk.peek().pos;
            history.push_back({end, start, 0});
            continue;
```

```
            }
    }

    // 首次处理该节点时，按启发式规则排序可能方向
    if (current.sorted_dirs.empty()) {
        struct MoveOption {
            int dir;
            int onward;
        };
        std::vector<MoveOption> options;
        for (int i = 0; i < 8; ++i) {
            int nx = cur_pos.x + dx[i];
            int ny = cur_pos.y + dy[i];
            if (nx >= 0 && nx < BOARD_SIZE && ny >= 0 && ny < BOARD_SIZE && board(nx, ny) == 0)
{
                int onward = count_onward_moves(board, nx, ny);
                options.push_back({i, onward});
            }
        }

        std::sort(options.begin(), options.end(), [](const MoveOption& a, const MoveOption& b)
{
            return a.onward < b.onward;
        });

        for (const auto& opt : options)
            current.sorted_dirs.push_back(opt.dir);
    }
```

1.2 GUI

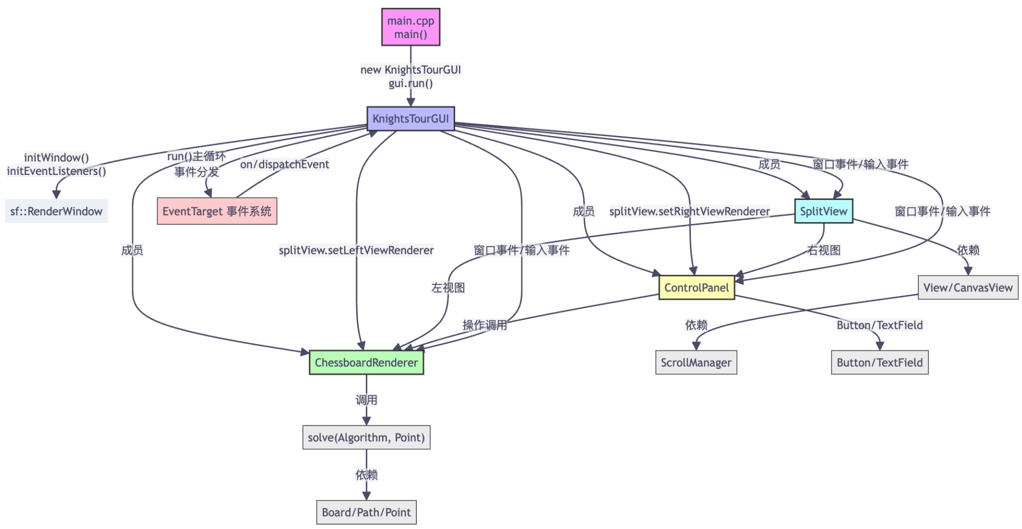


图 1　GUI 结构

使用第三方库 SFML

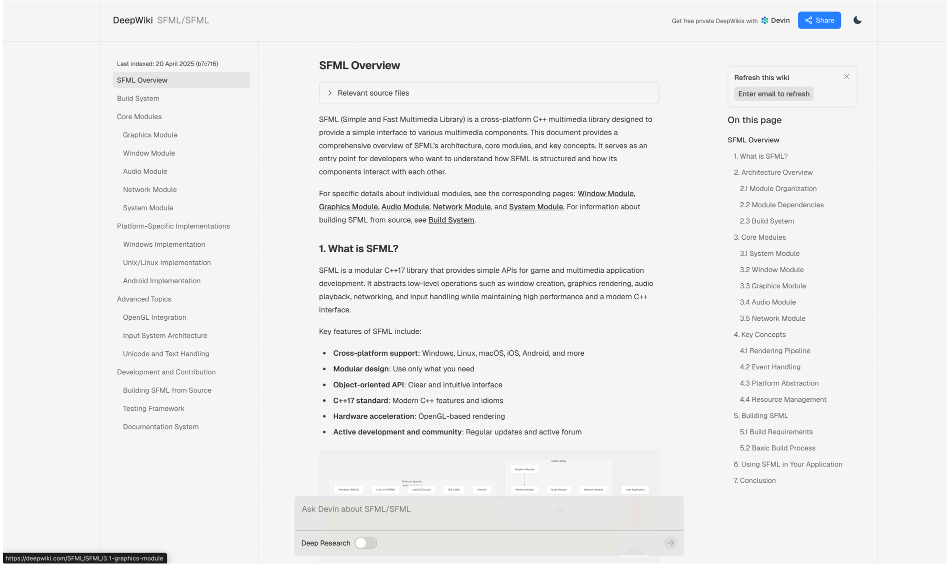


图 2 deepwiki

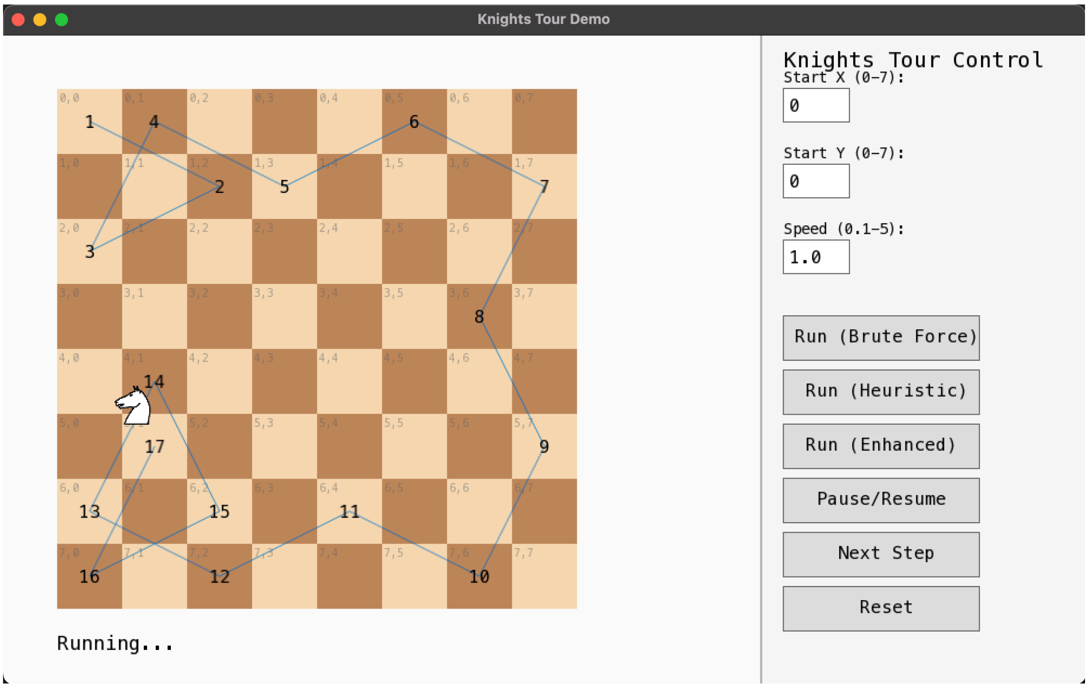


图 3 GUI 演示

二、 实习 6.4 平衡树操作演示

2.1 CLI

```
$ build/balanced_tree
commands:
    [q]uit
    [h]elp

    [c]reate <tree-id: a-z|A-Z> <algo: basic|avl|treap|splay>
    [d]elete <tree-id: a-z|A-Z>
    [p]rint <tree-id: a-z|A-Z>*

    [i]nsert <tree-id: a-z|A-Z> <key: int> <value: int>
    [r]emove <tree-id: a-z|A-Z> <key: int>
```

```
    [f]ind <tree-id: a-z|A-Z> <key: int>

    [s]plit <dest-id: a-z|A-Z> <src-id: a-z|A-Z> <key: int>
    [m]erge <dest-id: a-z|A-Z> <src-id: a-z|A-Z>

    [R]andom-insert <tree-id: a-z|A-Z> <count: int>
    [S]equential-insert <tree-id: a-z|A-Z> <start: int> <end: int>

trace mode:
    [n] or [\n]: next
    [c]: auto continue
>>>
```

提供了一个功能强大的解释器环境

支持创建/删除/输出树，在树中插入节点，删除节点，查找节点，分割树，合并树。

树名为单个字母 (a-z, A-Z)，可以使用任意算法 (basic: 普通 BST, avl: AVLTree, treap: Treap, splay: Splay) 创建树

支持随机插入和顺序插入若干个节点。

在每一个操作之后都会打印单步结果 (trace)，对树结构的任何操作（例如：连接或者断开连接子树）都会被实时记录下来。

显示 trace 时支持自动继续 (c) 和单步执行 (n)。

—

使用示例：

- 插入/删除/查找

```
>>> c A avl # create an AVLTree A
Created tree A with algorithm avl
>>> i A 1 10 # insert key-value 1-10 to tree A
Inserted {1: 10} into tree A
------------
Trace of tree A:
#1:
{1: 10}
>>> f A 1
Found {1: 10} in tree A
>>> i A 2 20
Inserted {2: 20} into tree A
------------
Trace of tree A:
#1:
    {2: 20}
{1: 10}
```

```
>>> i A 3 30 # insert cause imbalance, see
trace!
Inserted {3: 30} into tree A
------------
Trace of tree A:
#1:
        {3: 30}
    {2: 20}
{1: 10}
------------
(trace) c
#2:
{1: 10}
----
    {3: 30}
{2: 20}
------------
#3:
    {3: 30}
{2: 20}
    {1: 10}
```

—

```
>>> S A 4 12
Inserted sequential elements from 4 to 12 into tree A
...
>>> r A 6
Removed 6 from tree A
```

```
------------
Trace of tree A:
#1:
                {11: 92}
            {10: 25}
                {9: 100}
        {8: 28}
                {7: 84}
            {6: 71}
{4: 85}
            {3: 30}
        {2: 20}
            {1: 10}
----
{5: 5}
------------
(trace) c
```

—

```
#2:
            {11: 92}
        {10: 25}
            {9: 100}
    {8: 28}
        {6: 71}
{4: 85}
        {3: 30}
    {2: 20}
        {1: 10}
----
{5: 5}
----
{7: 84}
------------
```

—

```
#3:
                {11: 92}
            {10: 25}
                {9: 100}
        {8: 28}
            {6: 71}
{4: 85}
            {3: 30}
        {2: 20}
            {1: 10}
----
    {7: 84}
{5: 5}
------------
```

—

```
#4:
            {11: 92}
        {10: 25}
            {9: 100}
    {8: 28}
{4: 85}
        {3: 30}
    {2: 20}
        {1: 10}
----
    {7: 84}
{5: 5}
----
{6: 71}
------------
```

—

```
#5:
                {11: 92}
            {10: 25}
                {9: 100}
        {8: 28}
{4: 85}
            {3: 30}
        {2: 20}
            {1: 10}
----
    {7: 84}
{5: 5}
------------
```

—

```
#6:
            {11: 92}
        {10: 25}
            {9: 100}
    {8: 28}
            {7: 84}
        {5: 5}
{4: 85}
        {3: 30}
    {2: 20}
        {1: 10}
>>>
```

—

- 分裂：在$O(\log n)$ 时间内分裂出 $\geq$ key 的所有节点到一颗新树

```
>>> S A 10 20 # insert [10, 20) to tree A
...
>>> p
Tree A: AVLTree:
                {19: 116}
```

```
            {18: 139}
                {17: 173}
        {16: 187}
            {15: 159}
    {14: 143}
                {13: 169}
            {12: 160}
                {11: 195}
        {10: 124}
                {3: 30}
            {2: 20}
                {1: 10}
```

—

```
>>> s B A 15 # split tree A at key 15, result in tree B
...
>>> p
Tree A: AVLTree:
            {14: 143}
                {13: 169}
        {12: 160}
            {11: 195}
    {10: 124}
            {3: 30}
        {2: 20}
            {1: 10}
Tree B: AVLTree:
            {19: 116}
        {18: 139}
            {17: 173}
    {16: 187}
        {15: 159}
```

—

使用相同的命令管理不同算法的树

```
>>> c A avl
Created tree A with algorithm avl
>>> c T treap
Created tree T with algorithm treap
>>> c S splay
Created tree S with algorithm splay
>>> S A 1 10 # insert [1, 10) to tree A
...
>>> S T 1 10 # insert [1, 10) to tree T
...
>>> S S 1 10 # insert [1, 10) to tree S
...
>>> p ATS
Tree A: AVLTree:
                {9: 91}
            {8: 15}
                {7: 44}
        {6: 76}
            {5: 30}
    {4: 26}
            {3: 28}
        {2: 97}
```

```
            {1: 13}
Tree T: Treap:
                    {9: 33}
                {8: 98}
            {7: 51}
                {6: 94}
        {5: 67}
                {4: 94}
            {3: 98}
    {2: 21}
        {1: 61}
Tree S: SplayTree:
    {9: 14}
        {8: 27}
            {7: 92}
                {6: 47}
                    {5: 60}
                        {4: 13}
                            {3: 24}
                                {2: 62}
                                    {1: 54}
>>>
```

2.2 算法

语言：C++

使用 template 实现对于 `Key, Value` 的泛型支持

使用 CRTP pattern 实现代码复用

最大程度上复用代码的同时实现了多种平衡算法：`BasicTree`, `AVLTree`, `Treap`, `SplayTree`

三层类结构，逐级擦除类型信息

`TreeAdapter<K, V, Impl> : Tree<K, V> : TreeBase`

—

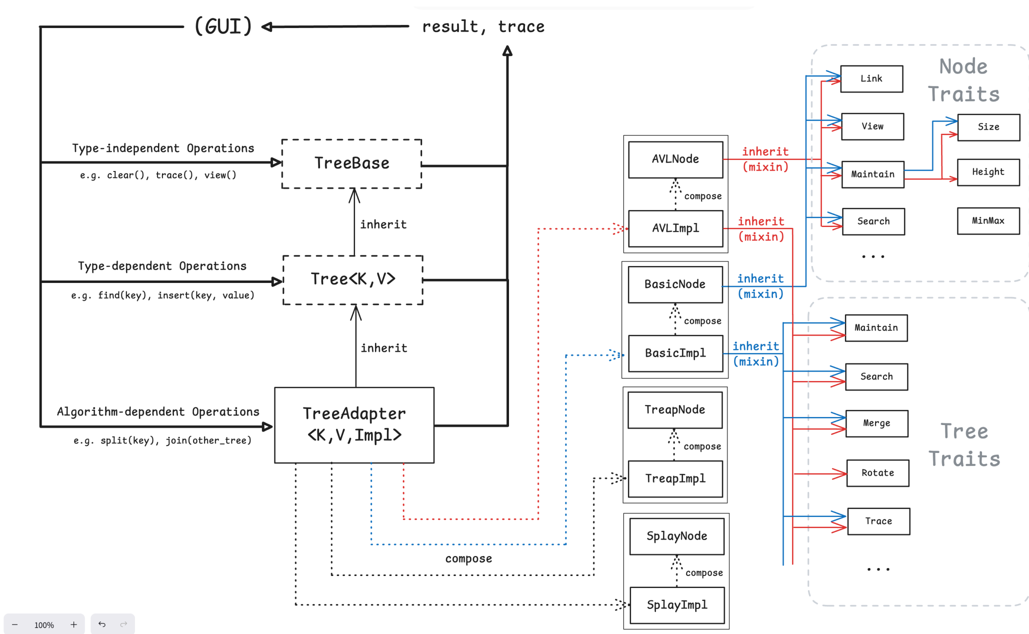


图 4 Tree 类图

—

2.2.1 基础操作实现

例：实现 find（trait::node::Search, trait::Search）

```
template <typename Node> struct Search {
    auto find(auto&& key) {
        // ...
    }
};
template <typename Tree> struct Search {
    auto find(auto&& key) {
        auto&& root = static_cast<const Tree*>(this)->root;
        return root ? root->find(key) : nullptr;
    }
};
struct BasicTreeImpl : Search<BasicTreeImpl>;
struct AVLTreeImpl : Search<AVLTreeImpl>;
struct TreapImpl : Search<TreapImpl>;
```

—

例：实现可提供不同功能的 maintain()（trait::node::Maintain）

```
template <typename Node> struct Height {
    int height{1};
    void maintain() {
        auto& self = *(static_cast<Node*>(this));
        auto l = self.child[L] ? self.child[L]->height : 0;
        auto r = self.child[R] ? self.child[R]->height : 0;
        self.height = 1 + std::max(l, r);
    }
};
template <typename Node> struct Size {
    size_t size{1};
    void maintain() {
        auto& self = *(static_cast<Node*>(this));
        auto l = self.child[L] ? self.child[L]->size : 0;
        auto r = self.child[R] ? self.child[R]->size : 0;
        self.size = 1 + l + r;
    }
};
```

—

```
// helper trait to maintain multiple properties
template <typename... Ts> struct Maintain : Ts... {
    void maintain() { (Ts::maintain(), ...); }
};

// imports a maintain() that maintains size
struct BasicNode : Maintain<Size<BasicNode>>;
// imports a maintain() that maintains both size and height
struct AVLNode : Maintain<Size<AVLNode>, Height<AVLNode>>;
```

—

2.2.2 旋转实现（`trait::Rotate`）

```
template <typename Tree> struct Rotate {
    void rotate(int dir, auto& root) {
        auto& self = *static_cast<Tree*>(this);
        auto new_root = self.unbind(root, dir ^ 1);
        if (new_root->child[dir]) {
            self.bind(root, dir ^ 1, self.unbind(new_root, dir));
        }
        auto parent = root->parent;
        self.bind(new_root, dir, std::move(root));
        self.moveNode(root, std::move(new_root), parent);
        root->child[dir]->maintain();
        root->maintain();
    }
    void rotateL(auto& root) { return rotate(L, root); }
    void rotateR(auto& root) { return rotate(R, root); }
    void rotateLR(auto& root) {
        rotateL(root->child[L]), rotateR(root);
    }
    void rotateRL(auto& root) {
        rotateR(root->child[R]), rotateL(root);
    }
};
```

2.2.3 AVL 树的 `split` 和 `join`（`AVLTree::{join, split}`）

- 先实现 `join(left_tree, sperator_node, right_tree)`：给定 key 值不交的两棵 AVL 树和一个 key 值在两树之间的分界点节点，合并成一棵树

  考虑 $\text{height}_{\text{left}} \geq \text{height}_{\text{right}}$ 的情况，反之对称

  在左树中找到高度为 h_{right} 或 $h_{\text{right}} + 1$ 的点 `cut_tree`，由于左树是 AVL 树，一定能找到

  将 `cut_tree` 和 `right_tree` 挂到 `seperator_node` 上，然后放回原先的位置

  高度最多改变 1，从 `cut_tree` 位置向上维护平衡即可。

  时间复杂度：$O(|h_{\text{left}} - h_{\text{right}}|)$

- `join(left_tree, right_tree)`:

  删除 `left_tree.max()` 或 `right_tree.min()`，转换为带 seperator 的 `join`

- `split(tree, key)`

  如右图所示，在 `find(key)` 的路径上的位置将节点和它的左右子树分开

  然后自底向上合并，每一次合并用路径中的点（图中的P_i）作为 seperator 合并两子树

$$\alpha P \longleftarrow \text{join}(\alpha, P)$$
$$\beta P_8 \beta_8 \longleftarrow \text{join}(\beta, P_8, \beta_8)$$
$$\alpha_7 P_7 \alpha P \longleftarrow \text{join}(\alpha_7, P_7, \alpha P)$$
$$\ldots \longleftarrow \ldots$$

  每一次合并的复杂度是高度差，高度差之和不超过总高度，所以复杂度为 $O(\log n)$

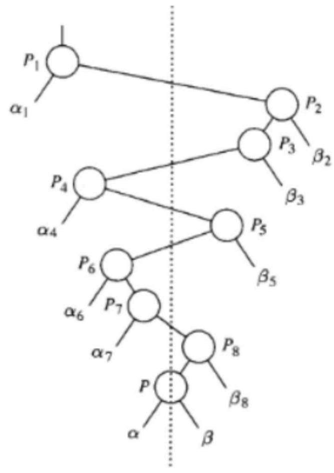


图 5　split 演示. ref.TAOCP

2.3　实现细节

```
struct TreeBase {
    virtual ~TreeBase() = default;
    virtual auto size() const -> size_t = 0;
    virtual void clear() = 0;
    virtual auto view() const -> ForestView = 0;
    virtual auto trace() -> std::vector<ForestView> = 0;
    virtual auto trace(const std::function<void()>& func) -> std::vector<ForestView> = 0;
    virtual void traceStart() = 0;
    virtual void traceStop() = 0;
    virtual void printCLI() const = 0;
    virtual auto stringify() const -> std::string = 0;
    virtual auto name() const -> std::string = 0;
};
```

—

```
template <typename K, typename V> struct Tree : TreeBase {
    virtual auto find(const K& key) -> Pair<const K, V>* = 0;
    virtual auto findKth(size_t rank) -> Pair<const K, V>* = 0;
    virtual auto min() -> Pair<const K, V>* = 0;
    virtual auto max() -> Pair<const K, V>* = 0;
    virtual auto insert(const K& key, const V& value) -> Status = 0;
    virtual auto remove(const K& key) -> Status = 0;
    virtual void traverse(const std::function<void(const K&, V&)>& func) = 0;
    virtual auto operator[](const K& key) -> V& = 0;
    virtual auto operator[](const K& key) const -> const V& = 0;
};
template <typename K, typename V, template <typename, typename> typename Impl>
struct TreeAdapter : Tree<K, V> {
    friend struct Test;
    auto size() const -> size_t override { return impl->size(); }
    auto view() const -> ForestView override { return impl->view(); }
    ...
    std::unique_ptr<Impl<K, V>> impl;
};
template <typename K, typename V>
using AVLTree = TreeAdapter<K, V, AVLTreeImpl>;
```

—

2.3.1 内存管理

使用 std::unique_ptr 管理节点所有权，防止内存泄露或者 double free 问题

2.3.2 Trace 记录 (trait::Trace)

- 在结构体中放一个 std::vector<ForestView> record; 记录每一步操作之后的状态

  所有对树结构的操作都通过调用 bind(), unbind() 方法，内部自动维护以及记录 trace
- 记录 trace 的方法:

  维护当前森林的根节点列表 std::set<Node*> entries;

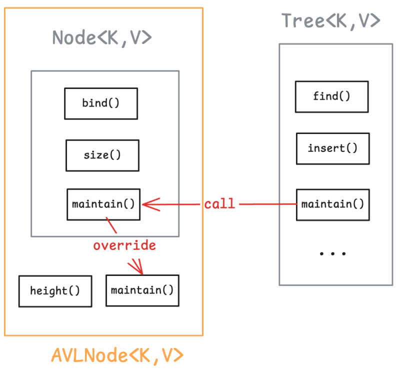
  每作一次记录 snapshot() 就复制出 entries 对应每一颗树中的信息，保存至 record

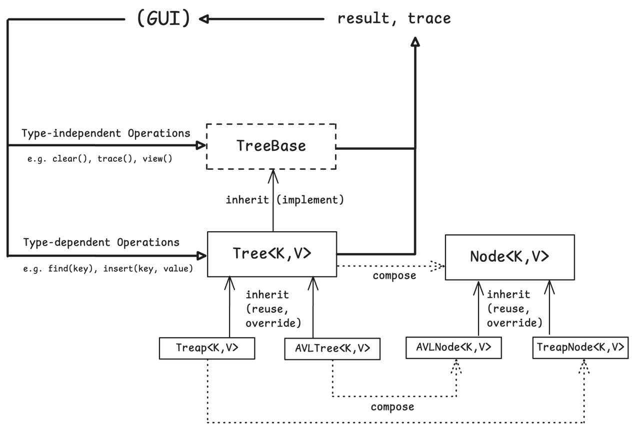
—

2.3.3 性能优化

最开始的结构:

- 非常容易想到





—

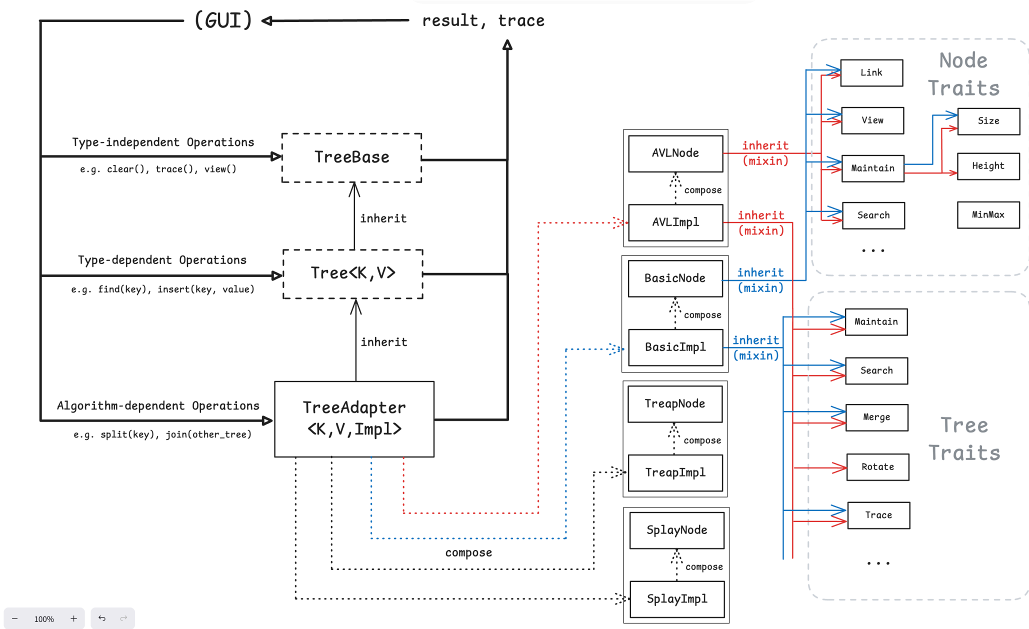
但

- 以 maintain() 为例，每一次自底向上维护信息时，都需要调用 node::maintain()，然而这是一个虚函数，但没法内联，每一次调用都有额外开销
- Tree 中只会存一个基类的 Node 指针，每一次使用子类 Node 特有信息时都需要 static_cast 或者 dynamic_cast
- 拓展功能比较麻烦
  - Splay 和 AVL 都可以旋转，要么实现两遍，要么创建一个 RotatableTree，增加继承层级
  - 更好的想法应该是把 Rotate 抽出来作为一个只提供旋转功能的 trait
  - 既然如此，为什么不把所有的功能都抽出来?

—

重构!

- 所有树平级，
  复用的功能只由 trait 提供
- 由于不同的树之间没有子类型关系，需要一个 TreeAdapter 来绑定到相同的接口上



—

查看是否内联:

- 重构前

```
$ nm | c++filt
000000010001e7c8 legacy::AVLTree<int, int>::AVLNode::maintain()
000000010001c4ec legacy::Tree<int, int>::Node::maintain()
```

  只有 `Tree::maintain()` 被内联了，Node 本身的 `maintain()` 没有被内联

- 重构后：

```
$ nm | c++filt
0000000100020578 AVLNode<int, int>::stringify() const
000000010001fe4c AVLNode<int, int>::~AVLNode()
```

Tree	Insert	Find	Remove
legacy::AVLTree(ms)	48.40	11.65	49.86
AVLTree(ms)	32.35	10.21	41.70
std::map(ms)	25.09	11.58	30.74
CRTP Improvement(%)	33.15	12.32	16.35

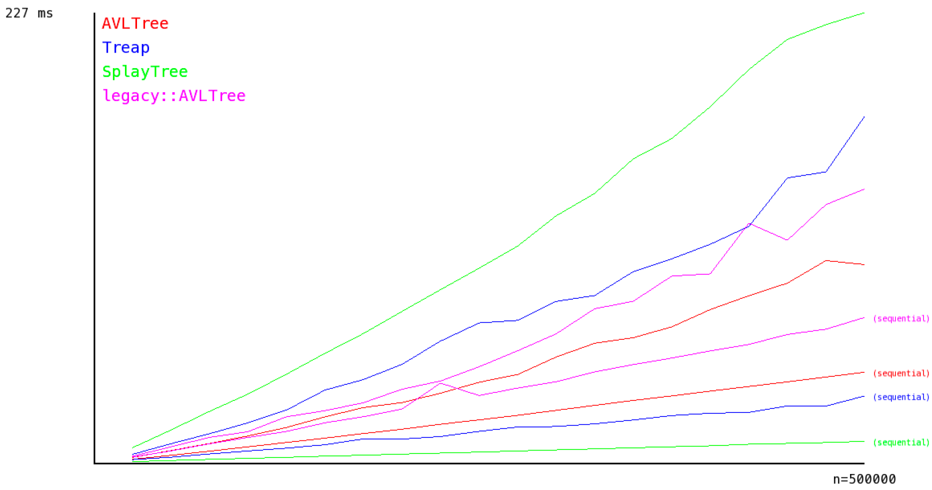


图 9　benchmark: insert

完整代码：https://github.com/cauphenuny/data-structure-assignment

2.4　单元测试

整个项目十分复杂，因此为每个功能写了测试

使用 `doctest` 库进行单元测试

```
$ build/balanced_tree test
[doctest] doctest version is "2.4.12"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:   26 |   26 passed | 0 failed | 0 skipped
[doctest] assertions: 6854 | 6854 passed | 0 failed |
[doctest] Status: SUCCESS!
```

测试举例

```cpp
SUBCASE("Split and merge") {
    // Split at 50
    auto other = tree->split(50);
    CHECK(other != nullptr);
    CHECK(tree->size() + other->size() == 7);

    Test::check(tree);
    Test::check(other);

    // Verify split worked correctly
    CHECK(tree->find(30) != nullptr);
    CHECK(tree->find(20) != nullptr);
    CHECK(tree->find(40) != nullptr);
    CHECK(tree->find(50) == nullptr);
    CHECK(tree->find(70) == nullptr);

    CHECK(other->find(50) != nullptr);
    CHECK(other->find(70) != nullptr);
    CHECK(other->find(60) != nullptr);
    CHECK(other->find(80) != nullptr);

    // Merge back
```

```
    tree->merge(std::move(other));
    CHECK(tree->size() == 7);
    CHECK(tree->find(50) != nullptr);
    CHECK(tree->find(70) != nullptr);
    CHECK(tree->find(60) != nullptr);
    CHECK(tree->find(80) != nullptr);

    Test::check(tree);
}
```